This is a post-peer-review, pre-copyedit version of a paper published in Silva F, Dutra I & Costa Santos V (eds.) Euro-Par 2014 Parallel Processing. Lecture Notes on Computer Science, 8632. Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, 25.08.2014-29.08.2014. Cham, Switzerland: Springer, pp. 415-426. The final authenticated version is available online at: https://doi.org/10.1007/978-3-319-09873-9_35

High-Performance Computer Algebra: A Hecke Algebra Case Study

Patrick Maier[1], Daria Livesey[2], Hans-Wolfgang Loidl[3], and Phil Trinder[1]

[1] School of Computing Science, University of Glasgow, Glasgow, UK
[2] School of Natural and Computing Sciences, University of Aberdeen, Aberdeen, UK
[3] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK

Abstract. We describe the first ever parallelisation of an algebraic computation at modern HPC scale. Our case study poses challenges typical of the domain: it is a multi-phase application with dynamic task creation and irregular parallelism over complex control and data structures.
Our starting point is a sequential algorithm for finding invariant bilinear forms in the representation theory of Hecke algebras, implemented in the GAP computational group theory system. After optimising the sequential code we develop a parallel algorithm that exploits the new skeleton-based SGP2 framework to parallelise the three most computationally-intensive phases. To this end we develop a new domain-specific skeleton, `parBufferTryReduce`. We report good parallel performance both on a commodity cluster and on a national HPC, delivering speedups up to 548 over the optimised sequential implementation on 1024 cores.

1 Introduction

Computational algebra is an important area of symbolic computation with many complex and expensive computations that would benefit from parallel execution. The area is served by a variety of systems, many specialising in some mathematical domain, for example GAP [7], a computational algebra system (CAS) specifically designed for group theory and combinatorics.

Some discrete mathematical problems are embarrassingly parallel, and this has been exploited for years even at Internet scale, e. g. the "Great Internet Mersenne Prime Search". Other problems have more complex coordination patterns and both parallel algorithms and parallel CAS implementations have been developed, e. g. ParGAP [5]. Many parallel algebraic computations exhibit high degrees of irregularity, at multiple levels, with numbers and sizes of tasks varying enormously (up to 5 orders of magnitude) [16]. They tend to use complex user-defined data structures, exhibit highly dynamic memory usage and complex control flow, often exploiting recursion. They make little, if any, use of floating-point operations.

This combination of characteristics means that symbolic computations are not well suited to conventional HPC paradigms with their emphasis on iteration over floating point arrays, and has motivated the development of scalable domain-specific scheduling and management frameworks like SymGrid-Par [16] and SymGridPar2 (SGP2) [20].

This paper outlines the first ever modern HPC-scale parallelisation of a problem in computational group theory, namely finding the invariant bilinear forms of Hecke algebra representations. These bilinear forms, and Hecke algebras more generally, are an

important tool in the study of symmetries that arise in many branches of mathematics, e. g. in topology and knot theory, with applications in theoretical physics and chemistry.

Our starting point is a sequential algorithm for computing bilinear forms, implemented in GAP. Prior to parallelising, we optimise the sequential algorithm, reducing sequential runtime by a factor of 350 (Section 2).[4] The paper makes the following research contributions.

(1) The development of a parallel algorithm for finding above bilinear forms. The parallelisation exploits the new SGP2 framework designed for scalable GAP computations. Core elements of SGP2 are a set of algorithmic skeletons, implemented in the parallel Haskell DSL HdpH [21], and a GAP binding for Haskell. We parallelise the three most time-consuming phases of the algorithm: (a) solving homomorphic images of linear systems over finite fields, (b) solving interpolation problems over rationals, and (c) bilinear invariance check (over polynomial matrices). All algebraic computations are performed by sequential GAP instances and coordinated by HdpH (Section 4).

(2) Some SGP2 skeletons are generic, e. g. the `parMap` parallel map of a function over a list. Other skeletons are specific to the algebraic domain. Specifically to compute with homomorphic images, a technique that is typical for a large class of algebraic algorithms, we have developed a new algebraic skeleton `parBufferTryReduce` that repeatedly checks whether the homomorphic results accumulated thus far are sufficient to reconstruct the final result (Section 3).

(3) Many mathematicians have access to commodity clusters rather than HPCs, so SGP2 is designed for both. We report good speedup and efficiency for a range of bilinear form problems, both on a Beowulf cluster and on medium-scale configurations of the HECToR UK supercomputer [12]. For example, one problem instance achieves a speedup of 548, coordinating 992 GAP instances on 1024 cores (Section 5).

2 Algorithm for Finding Invariant Bilinear Forms

Background. Using the terminology of [8], let $R = \mathbb{Z}[x, x^{-1}]$ be the ring of Laurent polynomials in an indeterminate x. For the purpose of this paper, it suffices to know that a *Hecke algebra*[5] $\mathcal{H}$ is an R-algebra with a basis $\{T_w \mid w \in W\}$ over R, where W is a finite Coxeter group with set of generators S. In this paper, we only consider Hecke algebras of type E_m ($m = 6, 7, 8$), that is, W is the exceptional Coxeter group E_m, and the cardinality of the set of generators S is m.

An n-dimensional *representation* ρ of a Hecke algebra $\mathcal{H}$ is an R-algebra homomorphism from $\mathcal{H}$ to $M_n(R)$, the R-algebra of $n \times n$ matrices over R. Note that ρ is *generated* by the matrices $\rho(T_s)$, $s \in S$. $\mathcal{H}$ is known to have a finite number of so-called *cell* representations ρ. Moreover, Howlett and Yin [13] have brought each of these cell representations ρ into a form where all m matrices $\rho(T_s)$ are sparse.

Graham and Lehrer [11] and Geck [8] show that for any given ρ there exists a non-trivial symmetric matrix $Q \in M_n(R)$, unique up to scalar multiplication, such that

$$Q \cdot \rho(T_s) = \rho(T_s)^{\mathsf{T}} \cdot Q \tag{1}$$

[4] Such dramatic optimisations are not unusual in computer algebra as the typical high-level presentation of computational mathematics often omits opportunities for sequential optimisation.

[5] More precisely, $\mathcal{H}$ is a one-parameter generic Iwahori-Hecke algebra.

for all generators $\rho(T_s)$. We call Q the *matrix of an invariant bilinear form*.

Depending on the representation ρ, finding the invariant bilinear form Q may require substantial computation. For each algebra type, the table below lists the number of cell representations ρ, the range of dimensions of ρ and the range of spreads of degree bounds of Laurent polynomials in Q. These numbers (and hence the difficulty of the problem) vary by several orders of magnitude.

Hecke algebra type	E_6	E_7	E_8
number of cell representations ρ	25	60	112
dimension of ρ	6–90	7–512	8–7168
spread of degree bounds of polynomials in Q	29–54	45–95	65–185

Sequential algorithm for computing Q. In principle, Q can be computed by viewing Equation (1) as a system of linear equations and solving for the entries of Q. However, solving linear systems over $\mathbb{Z}[x, x^{-1}]$ is too expensive to obtain solutions for high dimensional representations.

Instead, we solve the problem by interpolation. We view each entry of Q as a Laurent polynomial with $u-l+1$ unknown coefficients, where $u-l+1$ is the spread between lower degree bound l and upper degree bound u. Solving Equation (1) at $u-l+1$ data points will provide enough information to compute the unknown coefficients by solving linear systems over the rationals instead of $\mathbb{Z}[x, x^{-1}]$. To avoid computing with very large rational numbers (due to polynomials of high degree), we solve homomorphic images of Equation (1) modulo small primes and use the Chinese Remainder Theorem to recover the rational values.

The algorithm takes as input m generators $\rho(T_s)$ of dimension n, lower and upper degree bounds l and u, and a finite set of small primes P. From the degree bounds, we construct a set V_{lu} of $u-l+1$ small integers (excluding zero) to be used as data points for interpolation. The primes in P must be chosen large enough not to divide any of the integers in V_{lu}. The algorithm runs in three phases:

1. For all $p \in P$ and $v \in V_{lu}$, GENERATE a modular interpolated solution Q_{vp} of (1) by instantiating the unknown x with v and solving the resulting system modulo p.
2. For all $v \in V_{lu}$, REDUCE the modular matrices Q_{vp} by rational Chinese remaindering and obtain a rational interpolated solution Q_v of (1). Construct each Laurent polynomial q_{ij} in Q by gathering the (i, j)-entries of all Q_v and solving a rational linear system for the coefficients q_{ij}. Since Q is symmetric, there are $(n+1)n/2$ such systems, each of dimension $u-l+1$.
3. For all generators $\rho(T_s)$, CHECK that the resulting Q satisfies (1) over $\mathbb{Z}[x, x^{-1}]$.

After some (offline) pre-processing, the theory of Hecke algebras admits a particularly efficient way to GENERATE Q_{vp}. Instead of solving a linear system, the rows of Q_{vp} are computed by a *spinning basis algorithm* [9,17], multiplying, or *spinning*, the basis vector e of a pre-determined one-dimensional sub-space with n pre-determined products of the generators $\rho(T_s)$.

We observe that Q often has many identical entries. Therefore, the gather step of the REDUCE phase filters duplicates to avoid repeatedly solving the same linear systems. Typically, avoiding duplicates reduces the workload of REDUCE by a factor of 5 to 10.

Sequential optimisations. Profiling the GAP code on Hecke algebras of type E_6 lead to a number of improvements. The three most important ones are:

1. Avoiding unnecessary copying during the GENERATE phase by reducing the size of lambda abstractions encoding the generators.
2. Reducing the memory footprint by storing generators in a sparse matrix format.
3. Spinning the basis more efficiently by exploiting associativity.

For type E_6 these optimisations reduced sequential runtime of the algorithm (cumulative over all representations) by a factor of about 350, and the memory footprint by an order of magnitude from several GB to hundreds of MB.

3 The SymGridPar2 Framework

SGP2 system architecture. GAP [7] is the leading free system for computational discrete algebra. It is designed to be natural to use for mathematicians; to be powerful and flexible for experts and to be freely extensible so that it can encompass new mathematics. GAP supports very efficient linear algebra over small finite fields, multiple representations of groups, subgroups, cosets and different types of group elements, and backtrack search algorithms for permutation groups.

This case study used the most recent stable GAP distribution, GAP 4.6, which does not support parallelism. Hence the sequential GAP 4.6 instances are coordinated over the network by a distributed middleware, the SymGridPar2 (SGP2) framework [20]. The middleware occupies one core per multicore node and controls (via a RPC-like protocol) independent GAP 4.6 instances running on the remaining cores (Figure 1).

SGP2 itself is implemented in HdpH [21], a domain-specific language (DSL) for distributed-memory task parallelism, embedded in Haskell. SGP2 consists of two parts: (1) a GAP binding, enabling calls from HdpH to GAP, including automatic marshaling, and (2) a number of general-purpose and domain-specific parallel skeletons.

SGP2 programming model. HdpH is a monadic DSL, embedding a high-level coordination language into Haskell. Figure 2 introduces two central types of the HdpH DSL: `Par`, the monad type constructor for parallel computations, and `Closure`, the type constructor for serialisable values including unevaluated computations, or thunks.

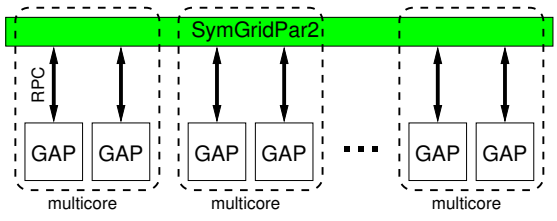

Fig. 1. SGP2 system architecture.

```
-- HdpH types
type Par a                                  -- parallel computation, returns result of type 'a'
type Closure a                              -- serialisable value/computation of type 'a'
type Task a = Closure (Par (Closure a))     -- serialisable parallel computation
                                            -- returning serialisable result of type 'a'

-- sample general-purpose skeletons
parMap    :: Closure (a → b) → [Closure a] → Par [Closure b]
parReduce :: Closure (a → a → a) → [Closure a] → Par (Closure a)

-- novel domain-specific skeleton; repeatedly reduces the results of a lazy list of input tasks
-- until the reducer computes a result
parBufferTryReduce :: ([Closure a] → Par (Maybe (Closure b)))  -- reducer
                   → Int                                       -- reducer batch size
                   → Int                                       -- number tasks eval'd in parallel
                   → [Task a]                                  -- lazy list of input tasks
                   → Par (Maybe (Closure b))                   -- result
```

Fig. 2. HdpH types and some SymGridPar2 skeleton signatures.

A `Task` is defined as a serialisable monadic computation returning a serialisable result. Thanks to serialisability, tasks and their results can be distributed over the network, and HdpH exploits this to provide automatic load management by work stealing.

At the lowest level, HdpH exposes fork/join style primitives for parallel programming. Using the primitives the HdpH library defines a number of general-purpose polymorphic skeletons (Figure 2), e. g. parallel maps (applying a function closure to a list of closures, in parallel) and reductions. The skeletons evaluate their input lists strictly as they coordinate monadic computations, and hence are unsuitable for computing with potentially infinite lazy lists.

Our case study requires solving an unknown number of subproblems in parallel until there are enough intermediate results to construct the solution. More specifically, the algorithm of Section 2 requires the use of an unknown number of primes in the GENERATE phase. A typical Haskell program would parametrise the GENERATE phase with an infinite lazy list of primes, and rely on demand from the REDUCE phase to decide how many primes are actually needed. As the monadic context of HdpH precludes lazy lists, we capture this domain-specific pattern[6] in a new skeleton that combines a task farm with a reducer.

The new `parBufferTryReduce` skeleton takes as input (in reverse order) a lazy list of tasks, the number of tasks to evaluate in parallel, the reducer batch size and the reducer function. A call to `parBufferTryReduce f b n tasks` continually forks from the head of list `tasks`, aiming to keep `n` tasks under evaluation, accumulating a list `accu` of intermediate results (not necessarily in the order of `tasks`). The reducer `f` is executed every time the length of `accu` is a multiple of the batch size `b`. The skeleton returns a result as soon as the reducer finds one; it returns `Nothing` only if the reducer fails to produce a result even after all `tasks` are evaluated.

[6] This pattern is common in algebraic computations that generate modular subproblems, e. g. linear system solving based on modular arithmetic and Chinese remaindering.

The HdpH DSL greatly simplifies developing domain-specific skeletons, particularly skeletons with complex parallel coordination such as `parBufferTryReduce`. A case in point is the implementation of the latter spanning less than 90 lines of code.

4 Parallel Algorithm for Finding Invariant Bilinear Forms

Each of the three phases of the sequential algorithm (Section 2) contains significant amounts of parallelism. Deciding what and how to parallelise is guided by the ratio between computation and communication costs on the distributed target architectures.

Parallel phases. Figure 3 shows the parallel structure of the algorithm to compute Q, with lower and upper degree bounds l and u, for an n-dimensional cell representation given by m generators $\rho(T_s)$; P is the set of primes used in the GENERATE phase.

The GENERATE *phase* forks $|P|(u-l+1)$ parallel tasks, each taking as input a pair of integers $(p, v) \in P \times V_{lu}$, where V_{lu} is defined as in Section 2. Each task runs the spinning basis algorithm to compute an $n \times n$ matrix Q_{vp} of integers modulo p. Thus the input size of GENERATE tasks is small and constant but the output size is quadratic in the dimension.

The REDUCE *phase* first constructs $k \leq (n+1)n/2$ interpolation problems by Chinese remaindering and filtering duplicates, then forks k parallel tasks solving the interpolation problems. Each task takes as input a vector of $u-l+1$ rational values, solves a linear system of $u-l+1$ equations over the rationals, and returns a vector of $u-l+1$ polynomial coefficients. Thus input and output size of REDUCE tasks depend (linearly) on the degree spread (and on the size of the rational numbers, which depends on the choice of P.)

The CHECK *phase* forks m parallel tasks, each checking the validity of Equation (1) w. r. t. one generator $\rho(T_s)$. To this end, each task requires as input the whole matrix Q, i. e. $(n+1)n/2$ polynomials with up to $u-l+1$ rational coefficients. Thus the input size of CHECK tasks is quadratic in the dimension and linear in the degree spread (and depends on the size of the rational coefficients), whereas the output is a single bit.

Overall coordination. Figure 3 depicts a parallel structure where REDUCE synchronises on the completion of GENERATE, which depends on the set of primes P. Instead,

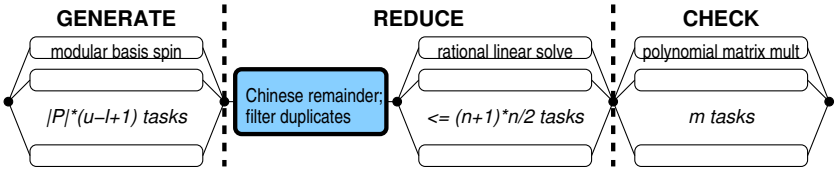

Fig. 3. Structure of parallel algorithm for computing invariant bilinear forms Q.

the `parBufferTryReduce` skeleton (Section 3) decouples GENERATE from REDUCE: The list `tasks` is a (possibly lazy and infinite) list of GENERATE tasks, the reducer `f` runs the REDUCE phase followed by the CHECK phase, and the batch size `b` determines the frequency of (attempted) reductions.

Note that most tasks in Figure 3 run on GAP workers and have a small memory footprint. However, the big task constructing the interpolation problems at the beginning of the REDUCE phase is executed on a dedicated GAP instance, the *GAP master*, because it must gather all Q_{vp} matrices and mangle them simultaneously, which may require substantial amounts of memory.

5 Evaluation of Parallel Performance

We evaluate the parallel algorithm (Section 4) on all cell representations (reps) for Hecke algebra of type E_7 and on the smaller reps of type E_8. The reps for type E_6 don't warrant parallel execution as their sequential runtimes are less than 150s. We evaluate on three different architectures:

- up to 16 nodes of a commodity cluster (Beowulf, 8 cores/node, 2.0GHz Intel Xeon CPUs, 12GB RAM/node, Gigabit Ethernet),
- up to 32 nodes of a Cray XE6 (HECToR [12], 32 cores/node, 2.3GHz AMD Interlagos CPUs, 32GB RAM/node, Cray Gemini interconnect), and
- a large memory NUMA server (Cantor, 48 cores, 2.8GHz AMD Opteron CPUs, 512GB RAM).

Figure 4 displays our results, organised into 2 columns: to the left data about the E_7 reps 3 to 60, to the right about the E_8 reps 3 to 16; reps 1 and 2 for E_7 resp. E_8 are trivial and easy to solve sequentially.

Problem size. The top row of Figure 4 plots the representations' dimensions and degree spreads (right y-axis) as well as the numbers of GENERATE and REDUCE tasks (left y-axis); recall that the number of CHECK tasks is constant at 7 and 8, respectively.

We observe that the number of GENERATE tasks tracks the degree spreads curve, whereas the number of REDUCE tasks oscillates by an order of magnitude or more though its trend is rising with the dimension.

To obtain reproducible results, the set of primes was chosen somewhat bigger than minimal, and the batch size parameter of the `parBufferTryReduce` skeleton was set so high that the reducer runs only once, after the GENERATE phase is completed.

Runtime. The second row of Figure 4 plots parallel runtimes, on 16 Beowulf nodes (using 15 * 7 + 1 = 106 GAP workers) in the case of E_7, and on Cantor (using 40 GAP workers) in the case of E_8. The graph for E_8 also plots the total work, i. e. the cumulative runtime of all tasks, and the time spent in the sequential part of the REDUCE phase. The graph for E_7 only plots the parallel work, i. e. the cumulative runtime of all

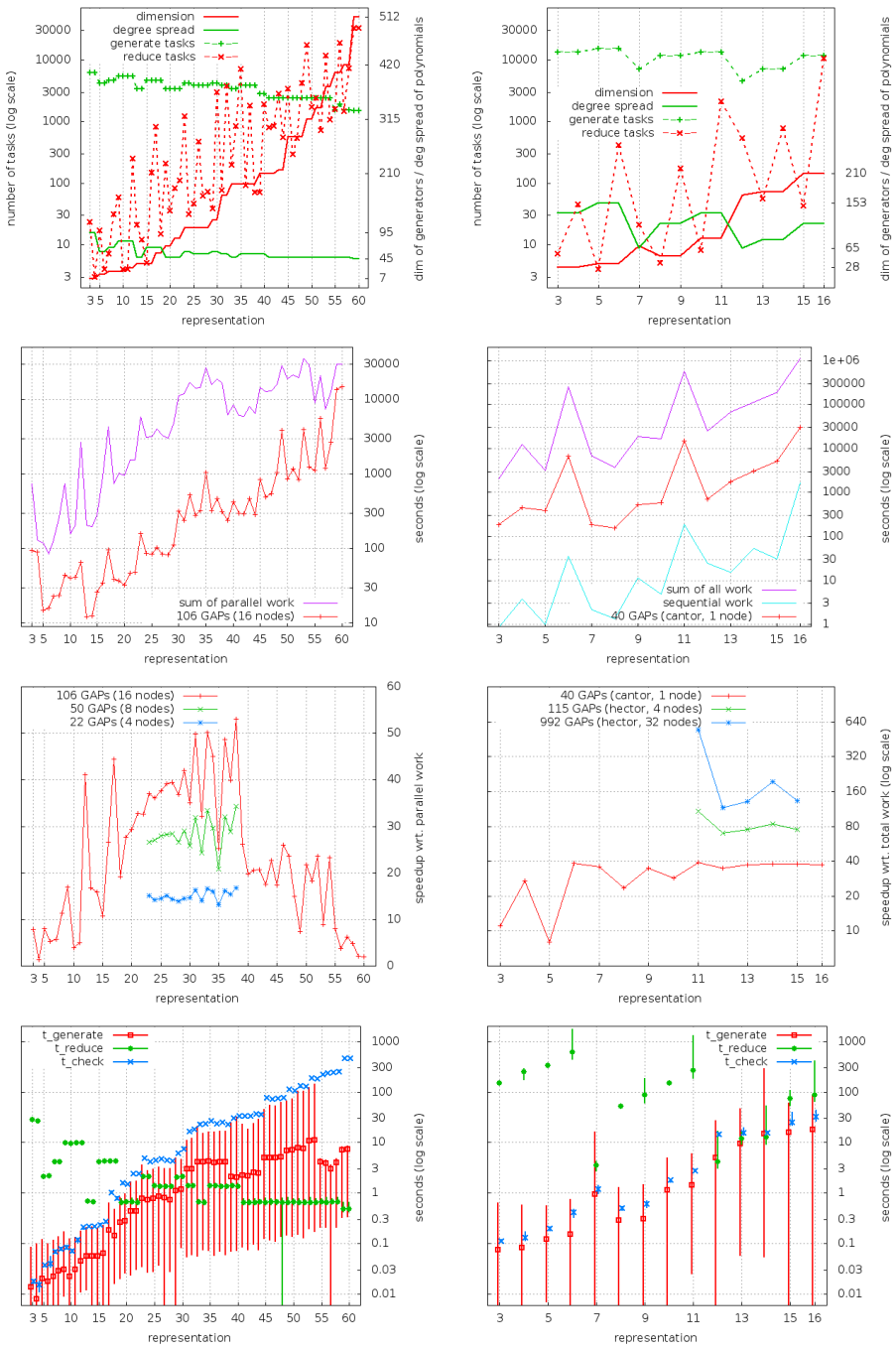

Fig. 4. Performance of parallel algorithm for finding invariant bilinear forms Q, E_7 to the left, E_8 to the right. Top to bottom: problem size, runtime, speedup, size of GAP tasks.

parallel tasks.[7] The reported times reflect single experiments as a statistically significant number of repetitions would be prohibitively expensive.

We observe that the amount of (total, parallel, sequential) work and the parallel runtime oscillate noisily due to the dramatic oscillation in the number of REDUCE tasks. The trend of work and runtime appears to grow with the dimension; the degree spread appears to have no influence.

Speedup. The third row of Figure 4 plots speedups on 16 Beowulf nodes (E_7, using 106 GAP workers) and on Cantor (E_8, using 40 GAP workers).

Since sequential runtimes are not available, we compute speedups w. r. t. parallel work (for E_7) or total work (for E_8). This method systematically underestimates the true speedup (particularly for E_7) as it fails to account for some of the costs of sequential execution, e. g. more time spent on sequential garbage collection.

We observe that most E_7 reps up to 22 are too small to produce significant speedups on 16 Beowulf nodes. Reps 39 and above, and particularly reps above 55, suffer from Amdahl's law due to significant time spent in the sequential part of REDUCE. Similarly, the E_8 reps up to 5 are too small for good speedups on Cantor. However, we cannot observe the effect of Amdahl's law for E_8; there is so much parallel work that speedups for reps 11 to 16 are close to the maximum of 40× despite rep 16 spending more than 1000 seconds in the sequential phase.

For the E_7 reps 23 to 38, we also investigate strong scaling from 4 to 8 to 16 Beowulf nodes. We observe that speedup oscillations increase with scale, i. e. some representations scale, others don't; best speedup (53×) is achieved for rep 38, corresponding to a best case efficiency of 50%. The picture is similar for the E_8 reps 11 to 15 when investigating strong scaling from 4 to 32 nodes on HECToR; rep 11 achieves the top speedup of 548×, top efficiency of 55%, but the other reps do not scale so well. Note that for multi-phase symbolic computations with irregular and dynamic parallelism an efficiency of 40% is good, as previously reported on smaller architectures [15,16,26].

Task size. The bottom row of Figure 4 shows the average, minimum and maximum runtimes of GENERATE, REDUCE and CHECK tasks; the time recorded is GAP compute time, excluding communication and marshaling overheads.[8]

We observe that CHECK tasks are generally expensive but regular, and REDUCE tasks are largely regular, with only some reps showing moderate irregularity (E_7 rep 48 is an outlier). However, GENERATE tasks are wildly irregular, varying by at least two orders of magnitude. The average cost of GENERATE and CHECK tasks appears to grow with the dimension, whereas the cost of REDUCE tasks appears to depend strongly on the degree spread.

[7] We failed to record the runtime of the sequential REDUCE step for E_7, thus can't provide total work; parallel work is an under-approximation.

[8] Overheads for calling GAP, including marshaling and data transfer, vary with task input and output size. For E_7 GENERATE tasks on Beowulf, for instance, overheads generally stay two orders of magnitude below average task size, ranging from about 10^{-4} to about 0.1 seconds.

Limitations. Two issues preclude solving the remaining E_8 reps with the current algorithm. First, the sequential time spent in the REDUCE phase, which grows quadratically with the dimension, obliterates speedups beyond dimension 200 (for E_7). The parallel algorithm needs to be redesigned to scale to dimensions between 1000 and 2000 (which are typical of E_8), let alone the maximum of 7168.

The second issue is the memory consumption, growing quadratically in the dimension, of the GAP master at the start of the REDUCE phase. The 12GB RAM of a Beowulf node prove insufficient already from E_8 rep 12, dimension 168.

6 Related Work

Computational algebra skeletons. This paper gives further evidence to the success of a parallel pattern, or skeleton, approach [2] in the domain of computational mathematics. We combine specialist domain knowledge, in the area of computational group theory, with language and systems knowledge, specifically for high-level orchestration of parallelism on large-scale clusters. This continues our work on domain-specific parallel patterns for symbolic computation, and some recent examples are as follows. We have designed a parallel Orbit, that achieves a speedup of up to 36 on a 64-core machine [14]; a critical-pair-completion pattern, with the Gröbner Bases computation as one instance that achieves a speedup of 6.9 on an 8-core machine; and the multiple-homomorphic images pattern, that achieves speedups of up to 11.9 on a 16-node cluster [18].

Parallel computational algebra. Several computer algebra systems offer dedicated support for parallelism (see [10, Sec 2.18] and [25]). Distributed Maple [26] provides a portable Java-based communication layer to permit interaction of Maple instances over a network. It uses future-based language constructs for synchronisation and communication, and has been used to parallelise several computational geometry algorithms. The Sugarbush [1] system is another distributed-memory extension of Maple, which uses Linda as coordination language. A distributed-memory parallel extension to GAP is the GAPMPI [3] package, which provides access to MPI functionality from within GAP. In contrast to this model of explicit message passing, our approach provides higher level abstractions, such as the `parBufferTryReduce` skeleton.

The TOP-C system provides task-oriented parallelism on top of a distributed shared-memory system [4], implementing several symbolic applications, including parallel computations over Hecke algebras [6] on networks of SPARC workstations.

Several efforts of parallelising computational algebra have targeted previous generations of HPC architectures. Sibert et al [27] describe the implementation of basic arithmetic over finite fields on a Connection Machine. Roch et al [24] discuss the implementation and performance of a parallel Gröbner basis algorithm on the Floating Point System hypercube Tesseract 20 with 16 nodes. Another parallel Gröbner basis algorithm is implemented on a Cray Y-MP by Neun and Melenek [23] and later on a Connection Machine by Loustaunau and Wang [19]. We are not aware of any other work within the last 20 years that targets HPC for computational algebra.

More recently main-stream computer algebra systems have developed interfaces for large-scale distribution, aiming to exploit Grid infrastructures [22]. The community

effort of defining a protocol for symbolic data exchange on such infrastructures allows interchange between different computer algebra systems [16]. In contrast to these Grid-based infrastructures, our technology targets massively parallel supercomputers.

Invariant bilinear forms for Hecke algebra representations. The invariant bilinear forms Q carry data that enables us to find so-called *Jantzen filtrations* [17], which simplify the general understanding of transformations of Hecke algebra representations.

Such bilinear forms Q for Hecke algebras of type E_7 and E_8 have previously been computed by Geck and Müller in an ad-hoc way; their paper [9] describes the mathematical basis for their approach but does not consider parallelism or evaluate performance. This paper and [17] are part of an ongoing project, started by Geck, to build a systematic GAP database of bilinear forms Q for Hecke algebras of type $\mathsf{E}_6, \mathsf{E}_7$ and E_8.

7 Conclusion

We have described what we believe is the first ever parallelisation of an algebraic computation on a modern HPC. The computation of invariant bilinear forms for Hecke algebra representations is multi-phase and exhibits irregular parallelism over the complex control and data structures typical of computer algebra. The parallelisation exploits the new skeleton-based SGP2 framework and required the development of a new domain-specific skeleton, `parBufferTryReduce`. The performance on a medium-scale HPC configuration and a commodity cluster is good, if noisy, reflecting the complexity of the problems solved. For example, for medium-size Hecke algebra representations (23 to 38) of type E_7 we obtain speedups of between 25 and 53 on 16 Beowulf nodes (128 cores, 106 GAP workers). For small E_8 representations (11 to 15) we obtain speedups of between 116 and 548 on 32 HECToR nodes (1024 cores, 992 GAP workers).

In related and ongoing work we report good performance results for small algebraic kernels on far larger HPC configurations, e. g. weak scaling of the `sumEuler` kernel (summing up Euler's φ function over large intervals) on up to 32K HECToR cores [20]. Core failures are predicted to rise along with the number of cores. To insure large and expensive symbolic computations against core failures, we have implemented and are evaluating automatic recovery of idempotent computations in SGP2 [28].

Acknowledgements. This research was supported by the grants HPC-GAP (EPSRC EP/G05553X), AJITPar (EPSRC EP/L000687/1), RELEASE (EU FP7-ICT 287510).

References

1. Char, B.W.: Progress report on a system for general-purpose parallel symbolic algebraic computation. In: ISSAC 1990, Tokyo, Japan. pp. 96–103. ACM Press (1990)
2. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
3. Cooperman, G.: GAP/MPI: Writing parallel programs in GAP easily. Tech. rep., Northeastern University, Boston, USA (1998)

4. Cooperman, G.: TOP-C: Task-oriented parallel C for distributed and shared memory. In: Workshop on Wide Area Networks and High Performance Computing, Essen, Germany. pp. 109–117. LNCS 249, Springer (1999)
5. Cooperman, G.: Parallel GAP: mature interactive parallel computing. In: Groups and Computation III, Columbus, OH, USA. pp. 123–138. De Gruyter (2001)
6. Cooperman, G., Tselman, M.: New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. In: ISSAC 1996, Zürich, Switzerland. pp. 155–160. ACM Press (1996)
7. GAP Group: GAP – groups, algorithms, and programming (2007), `http://www.gap-system.org`
8. Geck, M.: Hecke algebras of finite type are cellular. Invent. Math. 169, 501–517 (2007)
9. Geck, M., Müller, J.: James' conjecture for Hecke algebras of exceptional type, I. J. Algebra 321(11), 3274–3298 (2009)
10. Grabmeier, J., Kaltofen, E., Weispfenning, V.: Computer Algebra Handbook. Springer (2003)
11. Graham, J.J., Lehrer, G.I.: Cellular algebras. Invent. Math. 123, 1–34 (1996)
12. HECToR: UK National Supercomputing Service, `www.hector.ac.uk`
13. Howlett, R.B.: W-graphs for the irreducible representations of the Hecke algebras of type E_7 and E_8, private communication with J. Michel (December 2003)
14. Janjic, V., *et al*: Space exploration using parallel orbits. In: ParCo 2013, Munich, Germany. Advances in Parallel Computing, vol. 25, pp. 225–232. IOS Press (2014)
15. Konovalov, A., Linton, S.: Parallel computations in modular group algebras. In: PASCO 2010, Grenoble, France. pp. 141–149. ACM Press (2010)
16. Linton, S., *et al*: Easy composition of symbolic computation software using SCSCP. J. Symb. Comput. 49, 95–19 (Feb 2013)
17. Livesey, D.: High Performance Computations with Hecke Algebras: Bilinear Forms and Jantzen Filtrations. Ph.D. thesis, University of Aberdeen (2014)
18. Loidl, H.W., *et al*: Comparing parallel functional languages: Programming and performance. Higher-order and Symbolic Computation 16(3), 203–251 (2003)
19. Loustaunau, P., Wang, P.Y.: Towards efficient parallelizations of a computer algebra algorithm. In: Frontiers of Massively Parallel Computation, McLean, VA, USA. pp. 67–74. IEEE (1992)
20. Maier, P., Stewart, R., Trinder, P.W.: Reliable scalable symbolic computation: The design of SymGridPar2. Computer Languages, Systems & Structures 40(1), 19–35 (2014)
21. Maier, P., Trinder, P.: Implementing a high-level distributed-memory parallel Haskell in Haskell. In: IFL 2011, Lawrence, KS, USA. pp. 35–50. LNCS 7257, Springer (2012)
22. Maple Grid Computing Toolbox, `www.maplesoft.com/products/toolboxes/GridComputing`
23. Neun, W., Melenk, H.: Very large Gröbner basis calculations. In: CAP 1990, Ithaca, USA. pp. 89–99. LNCS 584, Springer (1992)
24. Roch, J.L., Sénéchaud, P., Françoise Siebert-Roch, F., Villard, G.: Computer algebra on MIMD machine. In: ISSAC 1998, Rome, Italy. pp. 423–439. LNCS 358, Springer (1989)
25. Roch, J.L., Villard, G.: Parallel computer algebra. Tech. rep., IMAG, France (1997), tutorial at ISSAC 1997
26. Schreiner, W., Mittermaier, C., Bosa, K.: Distributed Maple: parallel computer algebra in networked environments. J. Symb. Comput. 35(3), 305–347 (2003)
27. Sibert, E.E., Mattson, H.F., Jackson, P.: Finite field arithmetic using the Connection Machine. In: CAP 1990, Ithaca, USA. pp. 51–61. LNCS 584, Springer (1992)
28. Stewart, R.: Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell. Ph.D. thesis, Heriot-Watt University (2013)